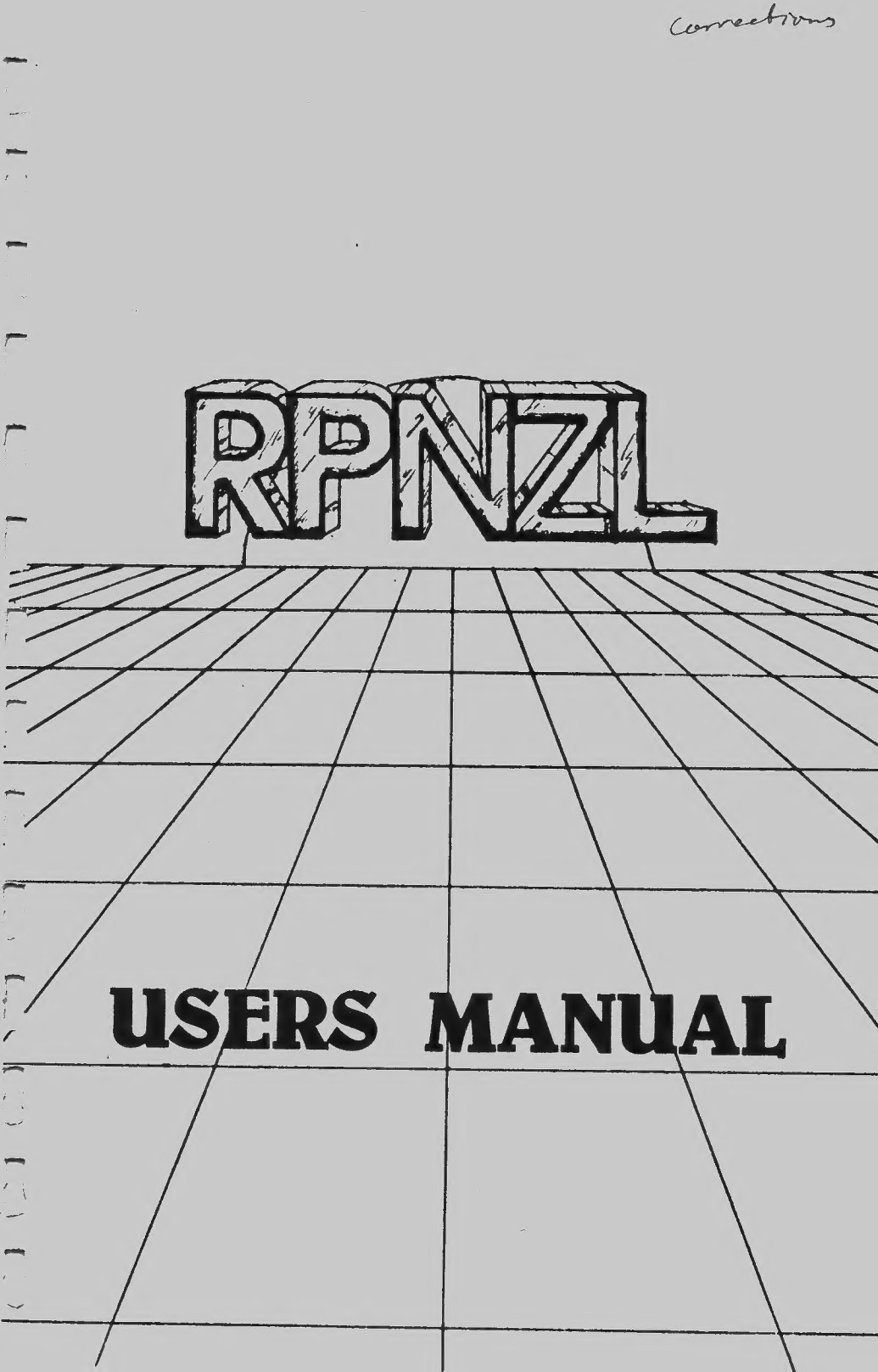
Corrections



USERS MANUAL

RPNZL™

PROGRAMMING SYSTEM

USERS MANUAL

for the Timex Computers

by Gary Shelden

Contents

INTRODUCTION.....1

I

Getting Started.....3
The Monitor.....3
RPNZL Tape System.....5
Loading a File.....6
Transfer.....7
Primary Tape Commands.....8
Tape Problems.....11
Running a Program.....12

II

Installing EDCOM.....14
Making a Text File.....14
Editor Functions.....15
Utility Commands.....15

III

Nature of RPNZL.....17
Programming Procedure.....19
User-Defined Words.....22
Subroutines & Global Variables.....25
Modules & Local Variables.....27
Program Segments.....29
Data.....30
Strings.....31
Numbers.....33
Function Keys.....34
Loops & If.....35

IV

Link File Structure.....41
Link & Run Addresses.....42

V
Stack Operators.....45
Memory Operators.....45
Arithmetic & Logic Operators.....46
Subroutine Operators.....48
Miscellaneous Operators.....49
Structure Operators.....49
String Operators.....51
Display Operators.....53
Input Operators.....56
Device Operators.....58
Tape Operators.....59
Array Operator.....61
APPENDICES
A Monitor & EDCOM Commands.....62
B Control Words.....63
C Errors.....64
D Translation from Sinclair BASIC.....66
E Memory Map and Sytem Variables.....67
F Machine Code.....69
G Code Summary.....70
Keychart.....Back Cover

INTRODUCTION

The RPNZL Programming System has been designed to provide an alternative environment for the ZX80, ZX81, TS1000, and TS1500 computers. This design was carried out with four primary purposes:

1) to increase the speed of program execution;

2) to reduce program and data memory requirements;

3) to give greater access to the hardware; and

4) to be simple enough to be hand-compiled (for development) without sacrificing the above features.

To accomplish these things, it was necessary to compromise in areas such as error detection, floating point functions, and interactive debugging.

RPNZL requires at least 16K of RAM; the system (variables, Interpreter, Monitor, etc.) occupies 9½K from 4Ø82. This leaves about 6½K, which, while seemingly small, is adequate for these reasons:

1) A RPNZL program is usually about half as long as the equivalent BASIC program;

2) RPNZL's integer variables occupy less than half the space of BASIC's;

3) portions of the system may be replaced by other programs; and

4) not all of a program's data needs to be in memory at one time.

The system should also work in a larger memory environment, although bit 15 of certain addresses is used as a flag by the Error and Function Key handlers.

This manual is designed not as a programming tutorial, but as an introduction to the use of the RPNZL language and operating system.

> **NOTE**
> Throughout this book, the ampersand & has been used to represent ENTER, the underscore _ to represent inverse type, and @ to represent the pound sterling symbol.

Chapter I THE MONITOR

GETTING STARTED

The System Tape is a regular BASIC program that contains the RPNZL language and Monitor within its first statement. It LOADs in a normal fashion and is self-starting. After loading, RPNZL is called and calculates a checksum of itself to insure that it has LOADed correctly. If the BASIC report appears with 8 on line 22 of the screen, this indicates that a LOAD error has occured. The program should be reLOADed. If further problems are encountered, consult the TAPE PROBLEMS section below. If the checksum was successful, the copyright notice will appear, followed by the RPNZL Monitor prompt ?>.

To exit RPNZL back to BASIC, type Q &. The BASIC report Ø/9 will appear, accompanied by the RPNZL report printed on line 22. When RPNZL is exited via Quit, the RPNZL report should be Ø, which indicates no error. To reenter RPNZL from BASIC, simply use RUN. RUN 5 will SAVE a self-starting copy of the System Tape.

THE MONITOR

The RPNZL Monitor is a program written in RPNZL executed immediately upon entering the system. It enables the user to manipulate programs and memory.

All Monitor Commands discussed below should only be used in response to the Monitor prompt ?>. The abbreviation "Cmd" will be used henceforth to mean "in response to the Monitor prompt, type the following Command:".

RUN RPNZL from BASIC. Cmd D0000 &. This Command line is expanded to DUMP 0000. The Monitor then dumps (displays) the bytes starting at the hex address 0000, 8 bytes per line, until a key is pressed.

Dump a line starting with 6500. It should be all 00s. Now Cmd W6500 &. The Command line is expanded to WRITE MEM 6500; 6500: appears below, and then the cursor with no prompt. Type 03 &; the byte is erased and repositioned beside 6500. Type 05 &; it is moved to the line above. Type & again; the Monitor prompt will reappear. Dump 6500 again. The line should read:

6500: 03 05 00 00 00 00 00 00

Cmd W6502 &; then A &. Note that A's code is written: 26. Type & again to return to the Monitor. Again Cmd W6502 &, then type shifted 9; the cursor's speed will increase, denoting Graphics Mode. Type H &. Note that & has cancelled the Graphics Mode. Type in the following characters, preceding each with a shifted 9: E & L & L & O &; then enter these bytes without the Graphics mode: 02 10 10 45 00. Dump 6500 again:

6500: 03 05 AD AA B1 B1 B4 02
6508: 10 10 45 00 00 00 00 00

Stop the dump and Cmd G6500 &. In the middle of the screen (approximately) should be printed HELLO.

The G(o To) Command causes the Monitor program to execute a RPNZL subroutine at the specified address. In this case, the subroutine would be written out:

$ "HELLO". 1010 >PRINT END

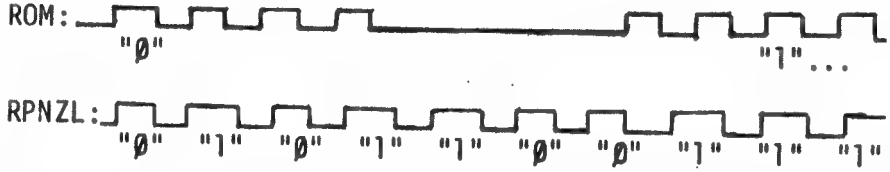
The $ indicates that a string follows, terminated by . (period); 1010 is the hexadecimal print position (column 10 hex, row 10 hex); >PRINT is like BASIC's PRINT AT.

Cmd M6500>6580,0010 &; then dump 6580; its contents should be the same as 6500. The M(ove) Command can move up to 0FFFF bytes in memory in any direction.

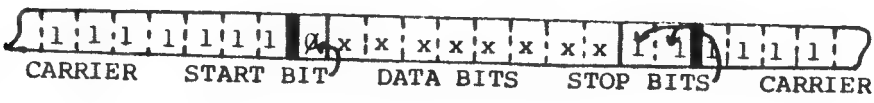
RPNZL TAPE SYSTEM

The ROM-based tape system is very slow; even though no error detection is employed, it is generally quite accurate. Each byte is recorded bit by bit; a 1 bit is represented by a tone burst twice as long as a Ø bit; and the spacing between bits is about as long as a 1 bit. While the frequency of the tone used is fairly high, using 4 cycles for a Ø and 9 for a 1 slows the data rate down to about 25Ø bits per second (bps). No system is used to mark where one byte begins and the other begins.

The RPNZL tape system uses approximately the same frequency as the ROM system, but instead of a tone burst to indicate a bit, a single pulse is used: a long pulse marks a 1 bit, and a short pulse a Ø bit.



This method increases the data rate to about 3ØØØ bps. In addition, RPNZL uses an "asynchronous" transmission scheme; rather than silence to indicate "no data" as the ROM uses, RPNZL uses a series of 1 bits, called the carrier. When a Ø bit is found in the stream of 1s, it is called a "start bit" and marks the start of a byte. That is, the next 8 bits received will constitute a byte. Following these 8 bits should be two 1 bits, called "stop bits". An error condition results if a Ø bit is found where a stop bit is expected. After the stop bits, the next byte may start immediately, or the stop bits of the last byte may merge into the carrier (continuous 1 bits), until a Ø bit is received marking the start of another byte.



In addition to the start/stop scheme (called framing), RPNZL uses a checksum; that is, all the bytes of the file (except the header) are added together, and the low order byte of this sum is included in the file header. When the file is read in again, the checksum is recalculated and compared to the checksum received in the header; an error condition results if the two checksum bytes are not equal.

The RPNZL tape system is more sensitive than the ROM system to variations in tape speed, to tape quality, and to tape head conditions. It is important that quality tape is used with clean heads on a reliable cassette player.

For this reason, all RPNZL applications are recorded in two formats: in the RPNZL tape file format, as described above, and also in a ROM-style format.

LOADING A FILE

Insert and rewind the reverse side of the ~~System Tape~~ Linker (marked SAMPLER). Cmd L and start the tape, but do not type & until the drive has wound the tape past its leader. Within ten seconds or so, the grey "fast" screen should show horizontal white streaks (each one is generated by a byte) in a manner very much unlike the ROM system. If the white streaks appear from the very beginning, the tape was either advanced too far, causing it to start mid-file, or the cassette speed is too different to be compatible. Try again, making sure that & is typed as soon as the leader is wound up. If it appears that the cassette speed is too different to be compatible, proceed to the TRANSFER section below.

When the screen returns to "slow", it should read:

```
?>LOAD
   SAMPLER P 6500 (len)(chksum)
   LOADED
```

If the BASIC report appears instead, there has been a Tape Error. On line 22 should be either a 6 or a 7. A 6 indicates a Framing Error (that is, when a stop bit was expected, a Ø bit was found); a 7 indicates a Checksum Error (that is, the file loaded but the calculated and received checksums did not match).

If the Load was unsuccessful, RUN RPNZL again. Using the three copies of the SAMPLER program try again to Load it. If after a few more tries, the SAMPLER still will not Load, proceed with the TRANSFER section below. If it has Loaded successfully, the following section may be read without doing what it says.

TRANSFER

Listen to the SAMPLER tape. The 3 copies of the RPNZL style file will appear as 3 noisy periods amidst the continuous howl of the carrier. After the RPNZL files will be a period of silence followed by two copies of the SAMPLER recorded in a ROM-style format. Set up the tape to start during the silent period. Be sure to readjust the volume to its normal Load level. Cmd TR & and start the tape. The program should Load. If additional difficulties are encountered, consult the TAPE PROBLEMS section below.

Rewind the SAMPLER to the beginning and Cmd S, and without typing & or any spaces, copy the SAMPLER header from the tape label, omitting the checksum given in parentheses at the end. The command line should now look like this:

```
?>SSAMPLER(P)65ØØ,nnnn
```

Disconnect the playback cable if necessary, and start the tape, still without typing &. Wait until the tape's leader has wound past, and then type &. The TV display will be similar to a ROM SAVE, but with much narrower bands. Stop the tape after the display reappears.

Rewind to the leader, start the tape in the play mode, and Cmd V, waiting until the leader is gone to type &.

The file should be visible as white streaks on the grey "fast" screen. When it has passed, the display should reappear and read:

```
?>VERIFY SAMPLER P 65ØØ nnnn nn
  OK
```

If it reports ERROR, try to Verify twice more. If it still will not Verify, Save it again, and then Verify. If it still reports ERROR, consult the TAPE PROBLEMS section below.

If Verification was successful, Save the SAMPLER twice more on the same tape. These three new copies made on the user's own machine should Load without problems on that machine.

TR (Transfer Read) always loads the file into memory at 65ØØ and places the file length in the 2 bytes at 64FE. Another Command, TWnnnn (Transfer Write), will record nnnn bytes in ROM-style format (for loading by TR Command) starting with address 65ØØ.

PRIMARY TAPE COMMANDS

FILE HEADER

Each file is written to tape with a header, which is a series of 15 bytes that transmit the file's name, type, address, size, and checksum. The tape Commands of the RPNZL Monitor require varying portions of this header to be included as operands. Below are listed the various components and their specific requirements:

fn$ - File Name - 8 characters or less; cannot include characters with code less than 1B hex (, and below).

t - File Type - 1 character only; 5 particular ones have been reserved:

P - Program File	T - Text File
D - Data File	L - Link File
S - Screen File	

aaaa - File Address - 4 hex characters
llll - File Length - 4 hex characters

The remaining header bytes, the SOF (start of file) and the checksum, are provided by the tape Commands.

SAVING A FILE

The S(ave) Command requires the complete header to be given:

Sfn$(t)aaaa,llll

If this format is not used exactly, the Monitor will ignore the Command.

Once the Command is accepted, the checksum is calculated, completing the header; then the carrier will be transmitted for about ten seconds, followed by the file header and file, and then about 3 seconds more of the carrier.

VERIFYING A FILE

A file should be V(erified) immediately after it has been Saved. Rewind the tape to the carrier before the file, start the tape and Cmd V &. If it returns with ERROR, try twice more. If it still will not Verify, Save it again and repeat the Verification. If Verify starts midfile, it will not work.

The header is not changed by Verify, so if problems are encountered lining up the tape to the carrier, the Command can be repeated. In fact, a file can be Saved several times (always a good procedure), and then all copies Verified consecutively.

If a file's contents in memory are changed between Save and Verify, an error will result. For this reason it is not possible to Verify Screen Files.

LOADING A FILE

The L(oad) Command has 3 possible operand formats:

1) L & - reads first file encountered into memory at address in SYSVAR FIL (4ØBE).

2) Lfn$(t) & - reads the file of name and type given into memory at address in SYSVAR FIL.

3) Lfn$(t)aaaa & - reads the file of name and type given into memory at address given; sets SYSVAR FIL to this address.

Headers of files that are encountered but do not match a named file are displayed in parentheses. If the Load Command specifies a file, the process will continue indefinitely until the desired file is found or until aborted.

Load operates without regard for the size or original address of a file; system crashes may occur if certain portions of memory are erroneously overwritten by Load.

FILE DIRECTORY

The F(ile Directory) Command reads and displays the headers of all files encountered until aborted by pressing any key. Between files a delay is executed to allow the tape start-up noise of the next file to be passed without creating an error condition.

AUTOSTART

The A(uto) Command reads the first file encountered into memory at the address pointed to by the SYSVAR FIL, and then Runs that file. Problems will probably arise if a file of a type other than Program is Run.

PICTURE

The P(icture) Command reads any file with the name and type SCREEN(S) into the Display File; it is then present when it returns to "slow". Typing & will cause the prompt to appear.

The Verify, File Directory, Auto, Picture and Transfer Read Commands do not require, and will ignore, any operands given with the Command.

ABORTING

Save cannot be aborted. Load, Verify, Auto, File Directory, and Picture may all be aborted by pressing any key.

FILE SPACING

If a file read begins mid-file, frequently a spurious header is found; it should be aborted immediately, or damage to other code may occur. Using the File Directory Command, however, eliminates this danger; it should be used to find the start of the file if there is any doubt.

When adding a file after another on tape, File Directory should also be used to locate the end of the last file. This may be done by letting the last file's name be displayed by File Directory, and then waiting until just before "fast" mode begins again before stopping the tape drive. This procedure is important and guarantees the file-to-file spacing that is essential for maintaining a large number of files on tape. It enables files in mid-tape to be overwritten with similarly sized files without disturbing surrounding files.

TAPE PROBLEMS

VOLUME

RPNZL files require the same volume level as ROM-style files; usually 3/4 maximum works best. Recorders vary a great deal, so experimentation may be necessary. The recorder's output should be 4 volts peak-to-peak at least; 5 or 6 volts is better.

HEADS

Clean the tape heads! This can be very important. Tiny bits of the tape's magnetic coating build up on the tape heads and rollers. This buildup can keep the tape from direct contact with the head, creating distortion. Occasionally the heads may require demagnetization or an adjustment of azimuth.

TAPE
Use a high quality, low noise tape. Any fingerprints or other dirt on the magnetic coating of the tape will probably render the tape useless, as well as spreading contaminants to the tape heads and thus to other tapes.

INTERFERENCE
Do not work on a metal table or near electrical appliances.

JACKS
It may be necessary to disconnect one cable while using the other; also be sure plugs are snug in their jacks. The jacks may wear out from frequent plugging and unplugging; they should be replaced with better jacks or tightened if this is a problem.

IF ALL ELSE FAILS
Get a new recorder. A $30 department store recorder should work fine. The RPNZL tape system requires a constant, dependable speed; some older recorders may not be able to provide this.

RUNNING A PROGRAM

Once the SAMPLER is successfully loaded, Cmd R6500 &. When the program ends, the prompt CS will appear in the lower left-hand corner of the screen.

The CS prompt indicates that the program being Run by the Monitor has ended. At this time, typing

> C will cause the 24 screen lines to be dumped to the printer; or
>
> S will cause the Display File to be Saved to tape under the header SCREEN(S)6069,0319. This Screen file may be reloaded using the Picture Command.

& will cause the screen to scroll, and the Monitor report and prompt will appear.

THE REPORT

After Running a program (with the Run or Auto Command) the Monitor gives a report, much like BASIC's report. The SAMPLER should end with the report:

RUN 65ØØ OK nnnn nnnn

The report is interpreted:

RUN 65ØØ The starting address of the program Run.

OK The error message; here indicates no errors have occurred.

nnnn The address of the last byte of the program executed.

nnnn The address of the Top Of the Stack. The Stack is where operands are put so that the operators can find them.

EXAMINE STACK

The SAMPLER program has ended with items on the Stack; that is, the address of the Top Of the Stack (TOS) is greater than 5B4Ø, which is the Stack Base.

Cmd X &. The Command line will be expanded to read:

?>EXAMINE STACK
STACK POINTER = nnnn

where nnnn is the TOS address as given in the report. Following will be a list of integers that are the contents of the Stack. The TOS integer is marked by <. Stop the listing by typing &.

Dump 5B4Ø; the dump should contain the same data as listed by the Examine Stack Command, except it is in a byte-by-byte format rather than integers.

Chapter II THE EDITOR

INSTALLING EDCOM

If the SAMPLER program was loadable in its original RPNZL-style format, the remaining applications provided probably should also load. If not, use the Transfer Read Command to load the ROM-style files; copy them using Save.

Rewind the EDITOR/COMPILER tape to its leader, start the tape and Cmd A & when the leader disappears. Be sure SYSVAR FIL (4ØBE) is set to 65ØØ before loading. EDCOM will install itself, providing 8 new Monitor Commands as listed.

MAKING A TEXT FILE

When installed, EDCOM creates an empty Text file. This file is simply an extended display file, consisting of 62 dec lines of 32 spaces each, each line bracketed by 76 hex ("newline"). The Editor uses the TV as a 24-line window on this file; the window may be moved up and down to view different parts.

Cmd E &. The Editor Menu will appear. Press any key and the screen will appear to clear with the cursor flashing in the upper left-hand corner. This position is called "Home". Type shifted A, then TEXT, and then shifted H(elp). Note that Help returns to the Editor Menu; below will appear the File Name that was just typed in. The grey square (shifted A) in the file is an Ø8 byte and indicates the length of the file name. It should not be omitted or replaced.

Return to the Text file by pressing any key. Enter a new file name, return to the Help screen, and observe that the name listed there is also changed.

EDITOR FUNCTIONS

Return to the file and practice typing. The four arrow keys, Delete and Graphics all work in a normal fashion. Enter acts as a carriage return. Attempting to move the cursor past the top or bottom of the screen will cause the "window" to scroll (if it can).

There are six additional Function Keys:

Shifted 1 - Insert Line - moves the first empty line found (starting at the top of the file) to between the cursor line and the following line, putting the cursor at the start of the new line.

Shifted 2 - Page Up - moves the "window" so that the cursor line is at the top of the screen, or as close as possible.

Shifted 3 - Clear Line - erases cursor line.

Shifted 4 - Quit - exits Editor back to Monitor.

Shifted H - displays Editor Menu without exiting Editor.

Shifted & - Home - moves "window" to the top of the file and places cursor in Home position.

UTILITY COMMANDS

There are five Monitor Commands installed with EDCOM that are prefixed with U(tility). These Commands overwrite the space in memory used by the TR and TW Commands. The Utility Commands are concerned with file manipulations.

UK - if Y is typed in response to the KILL FILE? prompt, the current text file is killed, that is, erased, and the Control Word STOP is placed at the end of the file.

UP - sends the current Text file to the printer. Operates regardless of OUTSTAT. May be terminated by typing &.

UST - saves the current Text file on tape as

fn$(T)72EØ,Ø81Ø.

This file may be Verified with the V Command.

ULT fn$ - file name must be 8 characters long including trailing spaces. Loads in Text file of that name if found, displaying the headers of all files as the L Command.

USL - saves the current Link file on tape as

fn$(L)7BØØ,Ø1FF.

May be verified with the V Command. Link files are the output of the Compiler and will be covered in detail later.

Chapter III THE COMPILER

NATURE OF RPNZL

NOTATION

RPNZL most resembles FORTH and RPL in that it uses Reverse Polish Notation (ZX-81 Language). Reverse Polish Notation, also called Polish Notation, was originally called Lukasiewicz Notation after its creator.

The word Notation refers to the method in which mathematical and arithmetical expressions are written.

BASIC uses Infix Notation; that is, the operators are within the expression and denote a relationship between the values before and after. For example, in the expression

```
LET A=2+3
```

the + denotes the relationship between the 2 and the 3, while the (LET)= denotes the relationship of 2+3 to A.

Reverse Polish Notation, on the other hand, places the operators after the operands; for example

```
2 3 + A LET=
```

is the exact Reverse Polish of the Basic expression above.

BASIC, while evaluating any expression, must turn its Infix Notation into Reverse Polish before it can calculate its value. This process of conversion uses a lot of time and contributes to BASIC's slowness.

In Reverse Polish Notation, however, the programmer effects this conversion as the program is written. On the surface this might seem laborious; however, in the expression

LET A=1/19*(23+B)

the programmer must decide the order in which the operations are to be performed, and, while observing a complex system of operator and function precedences, must decide how to structure the expression.

In Reverse Polish, the programmer still must decide the order of evaluation; the difference is that the expression is written just as it is to be evaluated. Thus, the example above would be written

1 23 B + 19 * / A LET=

BASIC's notation is closer to standard algebraic notation, but is not necessarily more natural.

THE STACK

Reverse Polish, as implemented both by BASIC's internal workings and by RPNZL's direct approach, requires the use of a stack.

A stack is a data structure in memory that acts like a stack of blocks, where any new blocks are added to the top; any blocks to be removed are also taken from the top. Thus, the first block stacked will be the last removed.

The numbers in a Reverse Polish expression are simply placed on the stack as they are encountered by the program. When an operator is encountered, it finds its operands on the Top Of the Stack (TOS), and places its results back on the stack.

The first Reverse Polish example above, 2 3 + A LET=, would cause the stack to go through these stages:

TOS→(empty) TOS→2 TOS→3 TOS→5 TOS→A TOS→(empty)

5

EXPRESSION: 3 + A LET=

Note that LET= is treated as an operator on par with + and A is treated as a value like 2 or 3 or 5.

NUMBERS

RPNZL uses hexadecimal integer arithmetic; that is, whole numbers only are allowed in the range Ø to 65535. This is the maximum range using 2-byte (16-bit) integers, and is generally adequate.

Floating Point numbers, as used by Sinclair BASIC, each require at least 5 bytes of storage, and more within programs. While Floating Point allows complex operations such as the trigonometric functions, the same complexity makes it slow.

Integer arithmetic, then, is much faster and saves a great deal of memory, but the tradeoff is the lack of complex functions. However, it is possible to write RPNZL code that will accomplish these things without slowing down all other operations as well.

PROGRAMMING PROCEDURE

OVERVIEW

RPNZL is a compiled language, that is, the program text (called the source code) is translated, or compiled, into the code the sytem can run (called object code).

The source code is contained in a Text file created by the Editor. The source code is then translated into object code by the Compiler. The object code is contained in Link files, which are finally joined together into a program by the Linker.

Since the Editor and Compiler (EDCOM) together allow only a 2K Text file and a ½K Link file, source code must be prepared modularly, that is, in pieces. EDCOM is then replaced by the Linker, which relocates itself to leave the program area of memory free. The Link

files are then loaded in one by one, and the compiled code segments are moved into position.

It is these segments that comprise a RPNZL program, much in the same way that lines comprise a BASIC program.

Segments in turn are comprised of "words". A word is a string of uninverted characters bracketed by characters whose codes are less than ØB hex; that is, the uninverted graphics characters are considered to be spaces, and spaces terminate words.

COMPILER STRUCTURE

A RPNZL program most simply consists of one Segment. This Segment may access the code of other Segments, which in turn may call on the code in still others.

The simplest Segment is the SUB(routine). Create a text file and type in:

```
■TEST
START 7DØØ
SEG SUB
$ "HELLO". 1Ø1Ø >PRINT
END
STOP
```

TEST is the file name; the grey square (shifted A) is actually a hex Ø8 and serves as the length marker for the file name string. It should not be omitted. The Compiler ignores the rest of the first line, leaving room for dates, titles, etc.

START nnnn tells the Compiler what address to start the compiled code at. SEG SUB declares that a subroutine code Segment follows. The next two lines should be familiar from the discussion of the Monitor above.

STOP actually is not necessary as the UK Command has already written it at the end of the file, but placing

it here keeps the Compiler from scanning all those empty lines. Note that when EDCOM is installed, STOP is not placed in the file; only the UK Command does that.

Exit the Editor and Cmd C(ompile) &. The ensuing Compiler output to the display is purely for the user to observe the program's progress; it is not identical with the Link file generated.

If the Segment was not entered correctly, the Compiler may report the error and stop. Typing & will automatically enter the Editor and Page Up to the line containing the error, where the word the Compiler was working on when it found the error has its first character inverted. If compilation was successful this time, go back and change the P in >PRINT to a B to force an error, then recompile and observe the error-handling procedure. The character marked should be restored when making corrections.

While the Compiler will detect certain errors, it cannot know the programmer's intent. Also, the errors that are detected are not always easy to identify: the word marked is only where the Compiler was when it found an error; it is not necessarily where the actual error lies. Appendix C should be consulted for suggestions on locating the culprit.

Once the sample program above is compiled, it must be linked. Since it would be inconvenient to erase EDCOM to bring in the Linker just to test a tiny Segment of code, EDCOM provides the (De)B(ug) Command. This Command links together the segments of a Link file into a test program. Cmd B, then Run. The program should run just like the sample discussed in Chapter 1. Using the E or C Commands destroys the run address set by the B Command; it may be restored by Running 5A8C.

Every program must be introduced by the Control Word START nnnn; every Text file must end with the Control Word STOP (included automatically by UK). Every Segment must be introduced by the Control Word SEG, and must have an implicit or explicit End.

COMMENTS

Comments in a RPNZL program are contained in ( ) and may be included in most Segment types. Misplacement of a closing parenthesis) may cause the Compiler to skip over sections of the source code.

USER-DEFINED WORDS

LABELS

In order to refer to a Segment from within another Segment, it is necessary to identify portions of code. BASIC uses line numbers, although it is possible to assign a line number value to a variable and then to use that as a label (e.g., LET GREET=1ØØØ GOSUB GREET). In RPNZL, labels are not variables, but can be treated as regular programming words or operators.

Labels are created by the Control Word DEF; their values are assigned in a variety of ways. In all cases DEF is followed by the label, which in turn is followed by an assignment operator(s), and finally by its value, which may be another operator. When the label is encountered by the Compiler within a Segment, the Compiler compiles the label's value into the code. There are three value types possible: a single byte, two bytes, or an Ø2 byte plus two bytes. This last type is the equivalent of simply stating a number, such as 1Ø1Ø in the program samples above; the code compiled simply puts the integer on the stack when run.

The three value types above may be obtained by using the following formats:

DEF label BYT nn creates a byte label;
DEF label WD nnnn creates a word label; and

DEF label PWD nnnn creates a push-word (Ø2+word) label.

Create a new file and set its START to 7DØØ and on the same line, type the Control Word FORGET. FORGET clears out the dictionary of old labels. If FORGET is forgotten, the dictionary may not be properly initialized and entries could overwrite other code.

Type the following program:

```
START 7DØØ FORGET
DEF BYTE BYT 8Ø
DEF WORD WD 1234
DEF INTEGER PWD 5678
SEG SUB
BYTE WORD INTEGER
END STOP
```

Cmd C and Cmd B, but do not run the program. Dump 7DØ8:

```
7DØ8:  8Ø 34 12 Ø2 78 56 ØØ
       BYTE WORD  INTEGER  END
```

The assignment operators WD and PWD require either a 4-digit hexadecimal number (nnnn) as above, or the operator HERE. HERE assigns the current compilation address as the label value.

Change the program above:

```
:
DEF WORD WD HERE
DEF INTEGER PWD HERE
:
```

Cmd C and Cmd B without running. Dump 7DØ8:

```
7DØ8:  8Ø Ø8 7D Ø2 Ø8 7D ØØ
```

There are two other assignment operators, CONS and VAR, that deal with the letter variables. These will be discussed in the SUBROUTINES section below.

DICTIONARY STRUCTURE

The dictionary of user labels consists of two strings; one holds all labels of 2 or 3 characters; the other, labels of 4 or more characters. Within each string, the labels, their types and their values are kept in this way: the label itself with bit 7 of its first character set, followed by a byte with a value of Ø, 1 or 2, then the value of the label, low order byte first. For example:

DEF LABEL PWD 1234

creates the entry:

B1 26 27 2A 31 ØØ 34 12

Label type Ø compiles an Ø2 byte followed by the value; type 1 compiles only the low order byte of the value; and type 2 compiles both bytes of the value.

Labels may be 2 or more characters in length; they may not be identical to any letter variable or operator; they may contain only characters with codes ØB through 3F.

Labels defined first are first in the dictionary string, and so are the first labels encountered when the Compiler searches for a label's entry. Since words of differing lengths are kept in the same string, if BAT were defined before BA, the Compiler would find a match for BA in the entry for BAT, with unhappy results. If the words are defined with BA first, the search algorithm will work.

Duplicate entries have no effect except to take up space; the first definition a label receives is the one used by the Compiler.

Since a string can only hold 255 characters, there is a limit to the number of labels that can be defined in a program. Even so, the capacities are, on the average, 45 words of 2 and 3 letters, and 30 words of 4, 5, and 6 letters; this seems generally adequate. When either dictionary string is full, compilation stops with the DICTIONARY FULL message.

SUBROUTINES & GLOBAL VARIABLES

SUBROUTINES

Create a new file and enter this program:

```
START 7DØØ FORGET
DEF GREETING PWD HERE
SEG SUB
$ "HELLO". PRINT END

DEF FAREWELL PWD HERE
SEG SUB
$ "GOODBYE". PRINT END

SEG SUB
GREETING PROC (PROCedure is like
                    GOSUB)
4Ø DELAY (like PAUSE)
FAREWELL PROC
END STOP
```

Cmd C, B, and R. Observe the results of the run. Note that Debug takes the last Segment as the principal (program) subroutine.

SEG SUB must be terminated by END. Omit one of the ENDs above and recompile to see the results.

GLOBAL VARIABLES

Change the last Segment above to:

```
:
SEG SUB
GREETING A " (pronounced "store")
FAREWELL B "
A @ (use the pound sterling symbol on
     the computer; this word is
     pronounced "fetch")
PROC 4Ø DELAY B @ PROC
END STOP
```

When compiled and run, the results should be the same as the earlier version.

Two new operators are introduced here, @ and ". The @ is used throughout this manual to represent the computer's pound sterling character. @ is pronounced "fetch" and replaces the TOS integer with the integer contained at that address. " places the Next On Stack (NOS) integer in the TOS address, removing both from the stack.

A and B are Global Variables; that is, they are accessible from all parts of a program. There are 52 Global Variables, A through Z and AA through ZZ.

Inspection of the compiled code shows that Global Variables are compiled as a single byte; that is, A is compiled as CC, B as CD, C as CE hex, etc. When run these bytes are converted into the actual addresses of the variables, using the SYSVAR GLOBVTBL as a base address. Debug sets this base to 7EØØ, so in the above program, A and B (compiled as CC and CD) are converted into the addresses 7EØØ and 7EØ2 on the stack.

In a full-sized program, the Linker determines the address of the Global Variable Table; in any program the address of this table is found in the 3d and 4th bytes of the program code.

The CONS assignment operator (used after DEF label) creates code that causes the Linker to place a certain value in a variable in the Global Variable Table. This variable may be changed by the program, but until it is, it is regarded as a CONS(tant). CONS requires a Global Variable plus a number as operands. The label is given a byte value, which happens to be the code byte for the letter variable.

For example, the above program could be changed:

```
        :
DEF GREETING CONS A HERE (HERE can be
                          used with CONS also)
        :
```

and in the last Segment, eliminate:

GREETING A "

and change the last line to:

GREETING @ PROC.

RPNZL provides a shortcut here: rather than A @ PROC, A FUN(ction) may be used to the same effect. This indirect subroutine call may be used in any situation where the address of the subroutine's address is left TOS.

Compile the modified program. Note that the B Command terminates with an error; this is because Debug cannot handle the code created by CONS.

MODULES & LOCAL VARIABLES

MODULES

In addition to the SUB Segment, there is another type of subroutine: the module, which is declared by SEG MOD and must be terminated by END.

The module acts as a single-byte subroutine call, as contrasted to the procedure, which requires 4 bytes (7D08 PROC is compiled to 02 08 7D 2D). Up to 50 dec modules, represented in source code by :00, :01, :02 ...:31 (compiled as 80, 81, 82...B1), can be declared.

When executed, the module byte is used to calculate an address within the MODTBL (set up by the Linker); at this address is an offset which, when added to the address of the MODTBL itself, points to the start of the module's code. The first two bytes of this code are the address of the module's Local Variable Table, much like bytes 3 and 4 of a program point to the Global Variable Table.

LOCAL VARIABLES

Each module may reserve up to 26 dec Local Variables (:A through :Z) for its exclusive use; they cannot be reached directly from the main program or from another module. They also are impermanent; that is,

they exist only during the module's execution and should not be used for permanent storage. However, if a PROCedure is called from within a module, then that PROCedure (or FUNction) is able to access the current module's Local Variables directly. Called from within a different module, the same PROCedure may access that module's Local Variables.

Local Variables used within the main program's code access the Global Variables A through Z. That is, the SYSVARs GLOBVTBL and LOCVTBL are identical while the program Segment is in control.

Local Variables may be used in DEF label VAR :1, but not in DEF label CONS 1 v.

DECLARING A MODULE

The module is declared by SEG MOD plus:

a) the module number, ØØ through 31. These must be assigned without leaving gaps (if ØØ and Ø3 are assigned, so must be Ø1 and Ø2) or a Linker error will result.

b) the level number, Ø through 7. The Global and Local Variables are assigned as a stack; the Global are on the bottom, with the succeeding levels of Local Variables on top. Levels serve only to permit nesting of modules without overlapping Local Variables (modules using the same level coincidentally share the same Local Variables); gaps in level assignment are permissible.

c) the number of Local Variables, Ø through 1A hex. The Linker assigns stack space based on the highest number of Local Variables declared for that particular level. It is thus possible to declare n Local Variables, and in error to use a variable higher than n: the results are unpredictable since the calculated variable address will fall outside the proper level.

For example:

```
SEG MOD ØØ 2 1Ø
SOME PROC
END
```

would compile a module using 16 dec variables :A through :P (although, perhaps it is really the PROCedure SOME that actually uses them); :ØØ (compiled to 8Ø) will call it as a subroutine. Note that any other module that uses level 2 will be using the same area of memory as this example.

It is generally desirable to define a label with the module's code (8Ø through B1) as a byte value:

```
DEF GREET BYT 8Ø
SEG MOD ØØ 2 Ø
$ "HELLO". PRINT
END
```

which could be called from elsewhere as:

```
SEG SUB
GREET 4Ø DELAY (etc.)
END
```

The Debug Command will not accept the code generated for modules, and will terminate with an error.

PROGRAM SEGMENTS

In order to link a complete program with the Linker, the final Link file must contain the code for a PROGram Segment; and therefore the last Text file of a program must contain a SEG PROG.

SEG PROG requires only one operand, a number Ø through 33 hex to declare the number of Global Variables. SEG PROG must be terminated by END.

SEG PROG will cause an error if used with the Debug Command, but a complete program cannot be linked without it.

DATA

It is occasionally necessary to include bytes of raw data within a Segment, or as a Segment itself.

Within a Segment, it is possible to coerce compilation of a byte of data by using a label with a byte value, or by using the operator DAT nn, where nn is the byte to be compiled. For example:

```
SEG SUB
$ "HELLO". DAT ØE PRINT
END
```

would compile code that would terminate in an error when run, because the ØE byte is an invalid code.

A Segment of data is declared by SEG DATA nn, where nn is the number of items to be compiled (in hex). Items may be either hex bytes (nn) or labels with / prefixed. For example:

```
DEF NEWLINE BYT 76
DEF DFILE WD 4ØØC
SEG DATA Ø5
21 /DFILE
36 /NEWLINE
C9
```

will compile into 21 ØC 4Ø 36 76 C9, which happens to be a short machine-code routine. This enables direct inclusion of machine code within a RPNZL program, and makes references to labels outside the machine code possible.

A label of type Ø, if used in a data Segment, will compile three bytes (Ø2 plus the value)--not generally a desirable thing to do.

Another type of data Segment is SEG BUF nn nn, where the first nn specifies the number of items to be compiled (like SEG DATA), and the second is the number of ØØ bytes to be reserved as a buffer (space for storage).

For example:

```
DEF TABLE CONS A HERE
SEG BUF Ø4 1E
Ø3 Ø3 Ø2 Ø5
```

will set up an array of 3 dimensions (Ø3, Ø2, Ø5) with 1E bytes, for a total of 1E+4=22 hex bytes. It also assigns the variable A the value of the array's address.

SEG DATA and SEG BUF do not require an explicit end word since their lengths are specified. These Segment types may be used with the Debug Command, and may contain comments.

STRINGS

The string in RPNZL exists in two formats:

a) in memory, as a byte representing the string length (Ø to 255 dec characters), followed by the actual bytes. This format is used in 3 situations:

1) as a string constant, it exists in memory as described above;
2) as a string variable (created by a running program), it is preceded by a byte representing the maximum length string the variable may hold, and is followed by string as described above.
3) as a string in compiled code, it is preceded by a Ø3 byte; when executed, the string is placed...

b) on the stack as a series of stacked integers, the lower bytes of which contain the string characters. The last character is on the bottom of the stack, and the first is NOS, with the string length TOS.

As has been seen previously, the string in source code is preceded by the operator $ (which must be followed by at least one space). The string must be terminated by a period . outside of quotes. Several other letters outside of quotes are used as formatting symbols:

C - compiles an EØ byte into the string; RPNZL treats this as a carriage return/line feed when printed, moving the print position to the start of the next line and scrolling if necessary.

Tnn - where nn is ØØ through 1F, compiles CØ through DF; RPNZL treats this as a tab when printed, moving the print position to the column indicated by nn on the same line. This is not the same as BASIC's use of TAB.

Q - compiles a quotation mark, much as BASIC's double quote symbol.

Nnn - compiles a byte of the value nn, regardless of its printability.

. - terminates the string.

Any other character outside of quotes (including graphics characters) other than spaces or a 76 byte (present in the Text file) will create an UNABLE TO COMPILE error with the $ inverted in the source code.

Within quotes, the characters are compiled exactly as found in the source code; however, a string that contains a 76 byte within its quotes will compile the byte as a space. For example:

```
$ CCC"NOW IS THE TIME FOR ALL GO
OD MEN".
```

will compile the next to the last word as "GO OD", while

```
$ CCC"NOW IS THE TIME FOR ALL "
"GOOD MEN".
```

would compile it as "GOOD".

Note that nothing is required to join the parts of the string as is the case of BASIC; the string terminator (period outside of quotes) takes care of this.

A string constant may be created in a program by using SEG $CONS followed by the string expression (without $). No explicit End is required beyond the regular string terminator. SEG $CONS may be used with the Debug Command.

Strings in memory are manipulated indirectly. In other words, in order to fetch or store a string (@$ or "$) to and from the stack, the address of the variable that holds the string's address (precisely, the address of the string's length) is placed TOS. This permits use of letter variables as well as arrays to refer to strings.

NUMBERS

As stated above, RPNZL uses hexadecimal integers. In source code they are represented in a variety of ways. As operands of DEF and SEG and their operators, the formats have been covered individually above; this section deals with numbers that are themselves words as in:

```
2 3 + A "
```

Since hexadecimal notation requires the use of letters as digits, it is necessary for the Compiler to have some way to distinguish numbers from labels, variables etc. This is done by requiring that all numbers begin with a character Ø through 9, as the following examples will illustrate:

```
Ø ØØ ØØØØØ Ø1 1 2 Ø2 ØC ØF 1D 1DF ØDF
1234 ØF123 ØFAB Ø9999 etc.
```

All numbers valued Ø, 1, or 2 are compiled to the bytes Ø7, Ø4, and Ø5, respectively. All numbers valued 3 through ØFF are compiled to single bytes preceded by an Ø1 byte. All other numbers are

compiled to low-byte, high-byte format preceded by a Ø2 byte.

The CH operator requires a single character on the same line set off by spaces as its operand; it compiles an Ø1 byte plus the character's code, just like a byte-sized integer.

When any of these compiled expressions is executed, the integer is placed on the stack as a two byte word.

FUNCTION KEYS

Within the RPNZL character table, used by the keyboard reading operators, the function keys are given values 4Ø through 4D hex (4E and 4F could be assigned but are not):

Shifted Enter	=	4Ø	Shifted	7	= 47
"	1	41	"	8	48
"	2	42	"	9	49
"	3	43	"	Ø	4A
"	4	44	Unshifted	Enter	4B
"	5	45	Shifted	Y	4C
"	6	46	"	H	4D

When INPUT decodes a character typed from the keyboard and a value 4Ø to 4F is found, the value is used to get a pointer from the table pointed to by the SYSVAR FKTBL. If bit 15 of this pointer from the table is set, a RPNZL subroutine is pointed to; if reset, it is a machine code routine. The appropriate subroutine is then executed from within the INPUT operator.

These subroutines are compiled by SEG USKEY (user key) which requires two operands: first, a single hex digit Ø through F determines the key for which the code is intended; and second, the operator SUB, which indicates a RPNZL subroutine follows, terminated by the .FUNK operator; or the operator CODE nn, which compiles nn data items just like SEG DATA.

When linked, the addresses of the various USKEY Segments are grouped into a table (with bit 15 of all RPNZL subroutine addresses set), and code is included at the beginning of the PROG Segment to set FKTBL to the new table, and at the end to restore the Monitor's Function Key table.

If one Function Key is redefined, the whole group , must be redefined; any left undefined are vectored to NOP.

The INPUT operator is terminated only by a machine code Function Key subroutine that returns with the Carry Flag set; if the table is redefined, at least one key will have to have a minimal machine code subroutine of 37 C9 (SCF RET), in order for there to be a way to terminate INPUT.

If a program terminates without restoring the Monitor's Function Key table pointer to FKTBL, as might happen with an error condition, it will probably be necessary to exit to BASIC in order to restore FKTBL.

LOOPS & IF

STRUCTURE

RPNZL is a fully structured language. This structure may be likened to the parentheses of a regular algebraic expression--for every opening parenthesis, there must be a closing parenthesis. Each opening, or beginning, structure must have as its counterpart a closing, or ending, structure.

Structure in RPNZL is the means by which "branching" decisions are made. Sinclair BASIC has only two branching facilities: FOR...NEXT and IF...THEN (with a statement, often GOTO, following). The BASIC approach can lead to problems in debugging since it is not always easy to backtrack a GOTO once it's gone. Structured programming reduces this problem

by leaving only one entrance and one exit to any particular section of code.

In order for branching/looping decisions to be made, it is necessary to evaluate true and false conditions. Technically speaking, "true" is ØFFFF, and Ø is "false". Practically, however, anything that is not false (not Ø) is therefore true. The branching operators of RPNZL base their decisions on the TOS integer. Operators like =, >, etc., return Ø or ØFFFF TOS, but any integer may be tested by a branching operator.

RPNZL has five types of branching/looping structures which will be discussed in the subsections below.

IF...ELSE....IF

IF causes the code between itself and ELSE, or between ELSE and .IF (pronounced "end if"), to be executed; the first clause if TOS was true (not Ø), or the second if it was Ø (false).

For example:

```
N @
IF
  $ "HELLO". PRINT
 ELSE
  $ "GOODBYE". PRINT
.IF
```

If N's value were Ø the GOODBYE would be printed, otherwise HELLO would. IF may be used without ELSE, but always requires .IF.

DO...LOOP

This structure is equivalent to BASIC's FOR...NEXT loop. DO expects the ending count (limit) TOS, and the starting count (Index) NOS. For example:

```
BASIC:   FOR I=Ø TO 1Ø
         :
         NEXT I

RPNZL:   Ø ØA DO...LOOP
```

STEPDO is like DO except that it expects a Step value on top of the Limit TOS. For example:

```
BASIC:    FOR I=Ø TO 1Ø STEP 2
          :
          NEXT I

RPNZL:    Ø ØA 2 STEPDO...LOOP
```

If LOOP finds that the Index equals or exceeds the Limit, the loop is terminated; otherwise the Index is increased by the value of Step (1 unless STEPDO has changed it), and then the loop is repeated.

The end of the loop may be forced by LEAVE, which simply copies the SYSVAR LIMIT into INDEX, where LOOP will find the terminal condition. Note that LEAVE does not actually terminate the loop; that is done by LOOP alone.

In BASIC, since a letter variable is used as the index for a FOR...NEXT loop, it is possible to use that variable directly within the loop. In RPNZL, this is done with the INDX operator, which copies the contents of SYSVAR INDEX to TOS.

BEGIN...UNTIL

There is no direct equivalent in Sinclair BASIC. BEGIN marks the start of the loop; UNTIL uses the TOS word to decide whether or not to return to BEGIN. If TOS is Ø, it loops back until UNTIL finds TOS not Ø. For example:

```
BEGIN
 KEY (KEY is like BASIC's INKEY$; if
      a key is pressed, the code for
      that key is returned, otherwise
      Ø)
UNTIL
```

will loop until a key (other than space) is pressed. The BASIC equivalent would have to use IF...THEN GOTO:

```
1Ø IF INKEY$="" THEN GOTO 1Ø
```

WHILE...THEN...WEND

There is no BASIC equivalent, but the WHILE loop is similar to the BEGIN loop, except that the branching decision is made at the start of the loop rather than at the end.

WHILE marks the beginning of the code that creates a value TOS to be tested by THEN. If THEN finds a Ø TOS it skips the loop and goes to the code following WEND. If TOS is not Ø, the code following THEN is executed, and WEND returns to WHILE to repeat the condition test and the loop.

For Example:

```
WHILE
 2 A @ * 4 =
 THEN
  $ "IT DOES". PRINT B FUN
WEND
```

would print IT DOES until B FUNction changes the value of A to 2.

SELECT...CASE...,CASE...,SLECT

This structure is a form of IF and is extremely useful. SELECT stores the TOS value in SYSVAR SLCT, and then CASE obtains the value and compares it to the value it finds TOS.

If the two values are equal, the code following CASE is executed up to .CASE (pronounced "end case"), and then it jumps to the code following .SLECT (pronounced "end select").

If the two values are not equal, the code bracketed by CASE...,CASE is skipped, and the process is repeated until a true case is found, or until .SLECT marks the end of the structure.

For example:

```
A @ SELECT
 1 CASE GREETING PROC .CASE
```

```
        2 CASE FAREWELL PROC .CASE
        B @ C @ + CASE DONOTHING PROC .CASE
        DOOTHERWISE PROC
      .SLECT
```

In the above segment, the value stored in variable A is tested against 1, then 2, and then B+C. If one case is found to match, that clause alone is executed. If no case matches, the code (if any) between the last .CASE and .SLECT is executed.

NESTING

The Compiler checks every loop for an appropriate ending structure for each beginning structure; a LOOP ERROR will occur if the loops are not balanced. Loops can be nested; for example:

```
SEG SUB A @
IF
   BEGIN
     Ø 4 DO
       SOME FUN
     LOOP
     B @
   UNTIL
  ELSE
   WHILE X FUN THEN
     Ø ØF 5 STEPDO
       OTHER PROC
       SELECT
         1 CASE Q FUN .CASE
         2 CASE Z FUN .CASE
         J FUN
       .SLECT
     LOOP
   WEND
.IF
END
```

Note the use of indentation to highlight the levels of nesting. The Compiler will allow about 7Ø dec levels of nesting before problems develop.

However, if while running a program, the loop stack, which holds return addresses, etc., growing downward, and the regular (parameter) stack, growing upward, meet, a NEST ERROR will halt execution. Omission of certain ending structures (or removal by code corruption) can also cause a NEST ERROR.

Chapter IV THE LINKER

LINK FILE STRUCTURE

As was seen above, each Segment of source code is compiled into object code in a Link file. Each code segment within the Link file is called a [redacted], and there are 6 types of records, A through F. Each type corresponds to one or more types of SEG.

Record Type	SEG Type
[redacted]	PROG
B	USKEY
C	[DEF label CONS...]
D	MOD
E	SUB, DATA, BUF, $CONS
F	[STOP]

The Linker will end with ERROR 1 if it finds a [redacted] with an illegal type. This usually indicates a corrupted file. The Debug Command recognizes only the record types E and F, and will terminate with an error if another type is found.

The Linker should be installed by Cmd LLINKER(P)65ØØ and then Cmd R65ØØ. Auto may be used if SYSVAR FIL contains 65ØØ. The Linker overwrites all of EDCOM: the Editor, the Compiler, and all the additional Commands are destroyed. They are replaced by the K (Linker) Command.

The K Command first queries the user NEW PROGRAM? (Y/N). A Y(es) response causes the Linker to clear the program area (65ØØ through 7AFF; 5½K), and to initialize its variables. Any other response leaves the program area and variables intact.

The Linker expects each Link file to be preceded by its Text file. When executed, the Linker skips a file (presumably, but not necessarily, Text), then loads the next file on the tape into 7BØØ. If it is not a Link file (type L), ERROR 1 will result. Note that a file longer than about 4½K read erroneously into 7BØØ stands a good chance of overwriting the machine stack, and any longer than ½K will overwrite the [redacted].

The [redacted] then processes each record very quickly and returns in time to find and skip over the next Text file. This process is repeated until a file containing a type A record (PROG) is read and linked, or until BREAK causes an exit from the [redacted].

When the PROG record is processed, the module, variable, and Function Key tables are set up as needed. Two other error conditions can arise at this time:

ERROR 2 indicates that there was a gap in the assignment of module numbers in the source code; and

ERROR 3 indicates that a C record (CONS) has declared a variable that falls beyond the Global Variable range assigned in the A (PROG) record.

LINK & RUN ADDRESSES

Every record contains the address for which it was compiled; the Linker reads the address from the first record of the first Link file only and assumes that all subsequent records will follow exactly. If the address in the first record of the first Link file is changed, then the Linker will be forced to link all the subsequent code at the new address, regardless of whether it will run. The advantage of this is that code can be compiled and linked for areas of memory in use by the Linker, and then moved into place.

The Compiler actually starts the object code at an address 8 bytes above that assigned by START. These 8 bytes are for: the address of the module table (bytes 1 and 2), the address of the Global Variable table (bytes 3 and 4), plus 4 bytes of code: Ø2 nn nn 2C. This code vectors the system to the actual program code (SEG PROG) at nnnn, which is almost at the end of the compiled code.

NOTE

The Linker's operation depends on accurate positioning of files as described on page 11; its activities are timed to locate the needed files as the entire series (Text/Link/Text/Link...) is played without user interference. Using the K Command, simply start the tape when prompted and the entire multi-file program will be linked from tape automatically. The program's addresses and size are output after the Linker is done.

Chapter V THE OPERATORS

RPNZL's instructions, what other languages call functions, keywords, commands, statements, etc., are called Operators. There are two groups: the first, those associated with the Control Words, was the primary subject of the previous Chapter; the second, a few members of which have appeared previously, are the actual instruction set of RPNZL and number 100. The purpose of this Chapter is to list and describe these 100 operators.

It should be reiterated here that RPNZL is stack-oriented; most of its operators find their operands and/or leave their results on the parameter stack. The first of the twelve categories discussed, Stack Operators, are the most unlike BASIC and yet the most useful; they concern themselves solely with the stack. Their primary use is to rearrange the order of integers on the stack, so that other operators can use them as operands.

The subsequent categories contain some operators that have names like BASIC's keywords; the functions are not always the same, however. Appendix D should be consulted to determine how best to render BASIC into RPNZL.

In the following discussion, it will be occasionally necessary to illustrate the parameter stack's contents; this will be done in this manner:

```
(1)-xx oper. (2)-nn
   yy            mm
```

where (1) and (2) point to the top of the stack (TOS) before and after the operator in question. The double letters serve only to represent two-byte integers.

STACK OPERATORS

SWAP DUPE DROP NTH ROT OVER SWITCH

SWAP replace TOS integer with NOS, and NOS integer with TOS.

```
(1)-xx SWAP (2)-yy
   yy          xx
   zz          zz
```

DUPE place a copy of TOS integer on top of TOS.

```
(1)-xx DUPE (2)-xx
   yy          xx
               yy
```

DROP remove TOS integer.

```
(1)-xx DROP (2)-yy
   yy
```

NTH remove TOS integer and call it n; place copy of nth deep integer TOS. Ø NTH returns copy of TOS (like DUPE); 1 NTH returns copy of NOS (like OVER).

```
(1)-Ø2 NTH  (2)-zz
   xx           xx
   yy           yy
   zz           zz
```

ROT remove TOS integer and call it n; remove nth deep integer and place it TOS. Ø NTH does not affect the stack; 1 ~~NTH~~ ROT swaps TOS and NOS.

```
(1)-Ø2 ROT  (2)-zz
   xx           xx
   yy           yy
   zz           aa
   aa
```

OVER place copy of NOS on top of TOS.

```
(1)-xx OVER (2)-yy
   yy          xx
   zz          yy
               zz
```

SWITCH swap high and low bytes of TOS integer.

```
(1)-hl SWITCH (2)-lh
   yy             yy
```

MEMORY OPERATORS

@ " PEEK POKE BUMP

@ (used here to represent the pound sterling symbol on the computer. Pronounced "fetch"). Replace TOS integer/address with integer contained at that address.

" (pronounced "store") Store NOS integer in TOS integer/address.

PEEK replace TOS integer/address with integer whose lower byte is contained at that address, and whose upper byte is ØØ.

POKE store lower byte of NOS integer in TOS integer/address. Upper byte of NOS is ignored.

BUMP increment by one the integer contained at TOS integer/address.

ARITHMETIC & LOGIC OPERATORS

+ - * / RSHIFT LSHIFT OVF NOT NEG AND OR XOR
= < > <= >=

+ ("plus") Remove TOS and NOS, add them and place sum TOS. If carry results store 1 in SYSVAR OVFW, otherwise store Ø in OVFW.

- ("minus") Remove TOS and NOS, subtract TOS from NOS and place difference TOS. Borrow sets OVFW to 1, no borrow sets it to Ø.

* ("times") Remove and multiply TOS and NOS, place least significant 16 bits of 32-bit product TOS, and store most significant 16 bits in OVFW.

/ ("divide") Remove TOS and NOS, divide NOS by TOS, place quotient TOS, and place remainder in OVFW. Division by Ø is allowed and returns ØFFFF.

RSHIFT shift all bits of TOS integer one place to the right, place a Ø bit in bit 15; set OVFW to 1 if least significant bit was 1, otherwise set OVFW to Ø. Same result as n 2 /, only much faster.

LSHIFT just as RSHIFT except TOS integer shifted left and least significant bit set to Ø. Same result as n 2 *, except much faster.

OVF ("overflow") Place contents of SYSVAR OVFW TOS. OVFW is affected only by the above six operators.

NOT remove TOS integer and replace it with its 1's complement, which is derived by inverting all bits.

NEG remove TOS integer and replace it with its 2's complement, which is derived by adding 1 to the 1's complement. Adding the 2's complement of an integer to n is the same as subtracting that integer from n.

AND remove TOS and NOS integers, and replace with their bit-wise AND, which is derived according to the table below.

OR remove TOS and NOS integers, and replace with their bit-wise OR, which is derived according to the table below.

XOR remove TOS and NOS integers, and replace with their bit-wise XOR, which is derived according to the table below.

X	Y	AND
Ø	Ø	Ø
Ø	1	Ø
1	Ø	Ø
1	1	1

X	Y	OR
Ø	Ø	Ø
Ø	1	1
1	Ø	1
1	1	1

X	Y	XOR
Ø	Ø	Ø
Ø	1	1
1	Ø	1
1	1	Ø

= remove TOS and NOS integers, return True (ØFFFF) if equal, False (Ø) if not.

< remove TOS and NOS integers, return True if NOS is less than TOS, False if not.

> remove TOS and NOS integers, return True if NOS is greater than TOS, False if not.

<= remove TOS and NOS integers, return True if NOS is less than or equal to TOS, False if not.

>= remove TOS and NOS integers, return True if NOS is greater than or equal to TOS, False if not.

SUBROUTINE OPERATORS

PROC CALL "CP FUN PROG END

PROC place contents of CPT on Loop Stack, remove TOS integer and execute RPNZL procedure at that address. When END is encountered, CPT is restored from the Loop Stack.

CALL remove TOS integer/address, and execute machine code at that address, terminated by RETurn.

"CP ("store Code Pointer") remove TOS integer and place in SYSVAR CPT. Used by Linker to vector to main program code.

FUN execute @ then PROC. Remove TOS integer/address, execute RPNZL procedure at the integer/address contained by that address.

PROG remove TOS integer/address, place into SYSVAR APPLIC, save contents of SYSVARs CPT, SPT, LSP, APPLIC, PSBOT, LSBOT, GLOBVTBL, LOCVTBL and MODTBL on the machine stack. Set SPT to 2Ø hex above PSBOT, make new LSBOT at LSP. Call RPNZL again as subroutine and execute code at APPLIC. Upon exit save CPT, SPT, and LSP into RCPT, RSPT, and RLSP respectively; restore nine SYSVARs from machine stack.

END if the Loop Stack is empty, exit current invocation of RPNZL by removing EXEC from machine stack. If Loop Stack is not empty, remove its TOS integer and store in LOCVTBL, and remove NOS integer from Loop Stack and store in CPT (thus effecting a return from subroutine).

MISCELLANEOUS OPERATORS

NOP RAN NEW DAT CH

NOP ("no op") does nothing

RAN remove TOS integer and replace with a pseudo-random integer between Ø and the TOS integer-1.

NEW reinitialize Parameter Stack by copying SYSVAR PSBOT into SPT.

DAT compile following byte of data into code.

CH compile following character's code into code as push byte operator.

STRUCTURE OPERATORS

IF...ELSE....IF BEGIN...UNTIL DO/STEPDO...LOOP
INDX LEAVE WHILE...THEN...WEND
SELECT...CASE....CASE...,SLECT

IF* remove TOS integer; if not zero, then execute code after IF. If zero, search following code for ELSE or .IF, and resume execution immediately after.

ELSE* search the following code for the corresponding .IF, and resume execution with code immediately following. ELSE is executed only if IF's condition was True, serving only as a marker for the false branch otherwise.

.IF ("end if") does nothing when executed, but serves as a marker for the false branch of IF, or for the exit of ELSE.

BEGIN place copy of CPT on the Loop Stack.

UNTIL remove TOS integer; if not Ø, drop BEGIN address from Loop Stack, and resume execution with code immediately following UNTIL. If TOS integer

is Ø, then copy integer from Loop Stack (BEGIN address) and place in CPT, causing jump back to code immediately following BEGIN.

DO place contents of SYSVARs LIMIT, INDEX and STP on Loop Stack, followed by contents of CPT. Set STP to 1. Remove TOS integer and store in LIMIT, remove NOS and store in INDEX.

STEPDO place LIMIT, INDEX, STP and CPT on Loop Stack as DO. Remove TOS integer and store in STP, remove NOS and store in LIMIT, remove third on stack and store in INDEX.

LOOP if the value of INDEX is equal to or greater than the value of LIMIT, drop CPT (address of DO) from Loop Stack, and restore previous loop's LIMIT, INDEX and STP; resume execution with code immediately following LOOP. If the value of INDEX is less than the value of LIMIT, add STP to INDEX and store in INDEX, and copy DO address from top of Loop Stack into CPT, causing jump back to code immediately following DO.

INDX place contents of SYSVAR INDEX TOS.

LEAVE copy contents of SYSVAR LIMIT into INDEX, causing termination of DO...LOOP upon execution of LOOP.

WHILE place copy of CPT on the Loop Stack.

THEN* remove TOS integer; if Ø, drop CPT from Loop Stack and search following code for the corresponding WEND and resume execution immediately after. If TOS is not Ø, continue with code following THEN.

WEND copy WHILE address from top of Loop Stack, causing jump back to code immediately following WHILE.

SELECT place contents of SYSVAR SLCT on Loop Stack, remove TOS and store in SLCT.

CASE* remove TOS integer and compare with contents of SLCT. If they are equal, execute the following code. If they are not equal then search following code for corresponding .CASE, and resume execution with code immediately after.

.CASE* ("end case") when executed, search following code for corresponding .SLECT; when found, restore previous value of SLCT from Loop Stack and resume execution with code immediately following .SLECT.

.SLECT ("end select") restore previous value of SLCT from Loop Stack.

> * These operators, ones which execute a "forward search" for their corresponding end structures, will terminate with NEST ERROR if END is found before an end structure. These operators do not check the nature of the end structure, so undetected errors can occur.

STRING OPERATORS

HEX DEC HEX$ DEC$ SWAP$ DUPE$ DROP$ AS$
@$ "$ +$ /$ =$ IN$

Throughout the discussion below, $ should be pronounced "string".

HEX remove TOS hexadecimal $ and replace with its integer value. If the $ is not 2 or 4 characters in length, terminate with $ EVAL ERR.

DEC remove TOS decimal $ and replace with its integer value. If the $ is longer than 5 characters, terminate with $ EVAL ERR.

> NOTE- the above two operators do not check the individual characters for value or type, so incorrect values can result.

HEX$ remove TOS integer and replace with 4-character hexadecimal $ TOS. Leading zeroes are included.

DEC$ remove TOS integer and replace with decimal $ equivalent TOS. Leading zeroes removed, so $ may have 1 to 5 characters.

SWAP$ replace TOS $ with NOS $, and NOS $ with TOS $.

DUPE$ place copy of TOS $ on top of TOS.

DROP$ remove and discard TOS $.

AS$ remove TOS and NOS integers, and assign a $ variable location at FREM NOS bytes long, and store address of $ variable in TOS address. Increment FREM by total size of $ variable (NOS+1), terminate with OUT OF MEM if FREM exceeds HIMEM.

@$ (@ is used to represent the pound sterling symbol; pronounced "fetch string") Remove TOS integer/ address and replace with $ pointed to by the contents of that address.

"$ ("store string") remove TOS integer/address. If the $ variable pointed to by the contents of that address is not large enough to hold the NOS $, make new $ variable as in AS$, placing new $ address into TOS address. Remove NOS $ and store in $ variable pointed to by the contents of the TOS address.

+$ ("plus string") remove TOS $ and NOS $ and concatenate with TOS $ attached to end of NOS $. If the resulting $ is longer than 255 characters, only the first 255 are returned. Place resulting $ TOS.

/$ ("divide string" or "slice string") remove TOS integer and call it TO, remove NOS integer and call it FROM, and remove third on stack $. Return portion of $ from FROM through TO (inclusive). If FROM is Ø or 1 the first character will be included. If FROM is greater than TO or $ length, a null $ (Ø) is returned.

If FROM equals TO, the resulting $ will be one character long. If TO is greater than $ length, the last character of the result will be the last character of the original $. Note that the TOS integer passed is the TO, and the NOS is the FROM, for example: $ "HELLO". 2 4 /$ will return $ "ELL"..

=$ remove TOS $ and NOS $. If they are identical, return True TOS, otherwise False.

IN$ remove TOS $ and NOS $. If TOS $ can be found within NOS $, return the integer byte position of the first match found; if not, return False. If the TOS $ is null, return True ($ by definition found).

DISPLAY OPERATORS

SEND PRINT >PRINT $PRINT CR POSN UP DOWN
CLEAR HOME CELL SHAPE DOT/UNDOT BIT FIG COPY

GENERAL

The main display driver, PRHAN, called by SEND, PRINT, >PRINT, $PRINT and SHAPE, checks for valid characters (replacing invalid ones with periods), and handles carriage returns (EØ) and tabs (CØ-DF), translating them into position information.

PRHAN also checks the OUTSTAT byte, which is used to redirect output to the printer or to a file.

After PRHAN handles each displayable byte, it calculates the next print position, scrolling up if necessary. Other than tab characters, which PRHAN handles itself, all characters sent to the printer driver in the ROM at address Ø851 (PNTA) are dealt with there.

SEND print lower byte of TOS integer in the current print position. Remove TOS integer.

PRINT remove and print TOS $ at the current print position.

>PRINT ("position print") execute POSN; that is, remove TOS integer and calculate new print position. Execute PRINT. Note that the POSN calculations do not affect the print buffer or printer.

$PRINT remove TOS integer/address, print string pointed to by contents of that address.

CR execute a carriage return. Same as ØEØ SEND or $ C. PRINT.

POSN remove TOS integer and calculate new print position using the high byte as the row number (Ø through 17 hex) and the lower byte as the column number (Ø through 1F hex). If the row number n is greater than 17 hex, the screen is scrolled up n-17 times.

UP scroll screen up, clear bottom line, and set the print position to 17ØØ, the first character of the bottom line.

DOWN scroll screen down, clear top line and execute HOME (set print position to ØØØØ).

CLEAR fill entire screen with spaces and execute HOME.

HOME set print position to home, that is, position ØØØØ, or Ø,Ø.

CELL execute POSN, then place byte found at print position TOS, as in PEEK.

SHAPE execute POSN. Remove NOS integer/address whose contents point to a $, the first byte of which determines the number of rows, and the second, the number of columns, in the Shape. Then starting at the print position, print the $ in n rows of n columns, tabbing to the original column position to start each row.

DOT/UNDOT remove TOS integer and calculate pixel position using upper byte as the Y coordinate (Ø through 31 hex) and the lower byte as the X coordinate (Ø through 3F), with the origin being in the lower left-hand corner of the screen. Note that this origin is two lines lower than that used by BASIC. Turn pixel on (black, or DOT) or off (white, or UNDOT). It is assumed that any position to be (UN)DOTted contains a valid plotting graphics character.

BIT calculate XY coordinates from TOS integer as in DOT/UNDOT, and return True if that pixel is on (black) or False if it is off (white).

FIG calculate XY coordinates from TOS integer as in DOT/UNDOT. (UN)DOT SYSVAR XSIZE bits per row of YSIZE bytes starting with NOS integer/address byte, for a maximum of 8 bits per row. This is the pixel equivalent of SHAPE. Note that NOS integer/address points directly to the series of bytes to be FIGured.

COPY dump contents of all 24 screen lines to the printer. Operates regardless of the condition of OUTSTAT.

OUTPUT REDIRECTION

This is controlled by the bits of the OUTSTAT byte at 4Ø7C. Bit 1 governs the printer, and bit 2 governs the output file. A 1 bit enables the device in question. If either bit 1 or bit 2 is enabled (set to 1), bit Ø is also set. These bits are manipulated by OPEN and CLOSE.

Bit 1 causes PRHAN to send each character to the printer driver (PNTA); carriage returns (EØ) are converted to 76 hex, and tabs (CØ through DF) reposition the printer buffer pointer (PR_CC).

Bit 2 causes PRHAN to place each character unchanged into memory, using SYSVAR FIL as a pointer, which is incremented each character.

INPUT OPERATORS

KEY CHAR INPUT >INPUT .FUNK

KEY scan keyboard and return code of shifted or unshifted key pressed. If unshifted space or no key at all is pressed, return Ø TOS.

CHAR get and debounce a single keystroke, shifted or unshifted. Similar to KEY, except CHAR waits for a kepress.

INPUT input and stack a string. Numerical inputs must be converted from string to integer form by HEX or DEC. The input cursor (see EDITING, below) appears at the current print position, anywhere on screen. The string typed as input remains on screen until overwritten or CLEARed. The SYSVAR MAXIN (4ØCB) governs the length of the input string, and has a maximum value of ØFF, as all strings.

>INPUT print TOS string at the current print position and then execute INPUT.

.FUNK ("end function key") terminates a RPNZL subroutine assigned to a Function Key. See the FUNCTION KEY section in Chapter III above. .FUNK actually removes the RPNZL return address EXEC from the machine stack, forcing a return to the program that called RPNZL, which in this case would have been INPUT. See Appendix F.

EDITING

Delete, right arrow and left arrow (shifted Ø, 8, and 5 respectively) manipulate the flashing cursor, but cannot leave the displayed string. If the cursor is positioned over an existing character, a new character typed will replace the original and the cursor will advance. Note that the ROM INPUT routine does not permit type-overs as it is always in insert mode. Enter, shifted or unshifted, terminates (>)INPUT.

Insert mode in RPNZL is toggled on and off by typing shifted 1. When insert mode is on, I will replace the character in the lower left corner (the "mode position") of the screen. When toggled off again, the character replaced is restored. If the cursor is at the right end of the input string while insert mode is on, typing a new character will toggle insert mode off.

If the right end of the string runs off the last line of the screen, automatic scrolling will take place provided that the new characters are not being inserted. The programmer should take care that the string is not pushed off the screen during insertion, as this may crash the display file.

If the input string includes the "mode position" and insertion is taking place, the I and the string might be visibly affected; however, the string returned to the stack is not affected.

BREAK KEY

The RPNZL BREAK key consists of unshifted space pressed simultaneously with the Ø key. This combination is scanned at the beginning of each instruction, provided bit 6 of INSTAT (see below) is on. CLOSE may be used to disable the BREAK facility.

INSTAT & INPUT REDIRECTION

The INSTAT byte at 4Ø7B controls input. Bits 3, 4, 5, and 7 control various aspects of the INPUT operator, such as Graphics, Insert, Flash, etc.

Bit 1 of INSTAT controls Input Redirection. If enable enabled, (>)INPUT gets bytes directly from memory, using the SYSVAR FIL as a pointer, which is incremented each time. INPUT is terminated by reaching MAXIN characters, or by 76 or EØ hex in the input stream. This provides an effective complement for Output Redirection to a file.

Bit 2 of INSTAT controls "continuous input". In this mode, whenever MAXIN is reached, the input buffer is emptied and the pointers reset, allowing the (>)INPUT operators to continue endlessly until a Function Key returns with Carry Set. If the last character position on the screen is reached, Function Key ØB is automatically executed. With the Function Keys as provided by the system, this would simply terminate the operator. The string returned by this mode consists only of the characters typed since MAXIN was last reached and the buffer emptied. As this is usually garbage, it should be DROP$ed.

Bit 6 of INSTAT governs the operation of the BREAK combination as seen above.

KEYBOARD LAYOUT

Since RPNZL does not make use of the keyword entry system used by BASIC, it has been possible to rearrange the keyboard layout slightly with regard to the graphics characters. The chart on the back cover will provide specific details. The graphics characters have all been grouped together on the QWERT and ASDFG keys. This arrangement permits the use of the same key table in regular or graphics mode; ALL graphics mode characters are the exact inverse (shifted or unshifted) of the regular mode characters.

DEVICE OPERATORS

OPEN CLOSE FILE

As has been seen in the two sections above, the INSTAT and OUTSTAT bytes control in/out redirection as well as other functions. OPEN and CLOSE are used to set and reset, respectively, the bits of the two bytes. Both operators use the lower byte of the TOS integer; if bit 7 is set, the operation is performed on INSTAT, otherwise on OUTSTAT. One other bit of the integer is set; it is the NEXT HIGHER bit position that is toggled by OPEN and CLOSE.

Each bit so manipulated represents a logical "device" for the I/O operators. The devices and their operands for use with OPEN and CLOSE are:

INSTAT (4Ø7B)		
File input	Bit 1	81
Cont. input	Bit 2	82
Break Key	Bit 6	ØAØ
OUTSTAT (4Ø7C)		
Printer	Bit 1	Ø1
File	Bit 2	Ø2

OPEN turn on "device" indicated by TOS integer.

CLOSE turn off indicated "device".

FILE remove TOS integer and store in SYSVAR FIL. FIL is used by the input and output operators for file I/O and by the tape operators.

TAPE OPERATORS

WRITE READ VER

WRITE expects a series of ØE hex bytes as integers TOS to form the File Header. There should be no string length byte. In the following illustration,

SOF indicates Start Of File. This signals READ and VER that a valid file is starting.

TYPE may theoretically be any byte; however, the Monitor has made certain assignments regarding type. It should be noted that the Monitor's Save routine subtracts 1C hex from the code of the character given as type; thus file type P is stored as a type byte 19, and file type Ø as type byte ØØ.

NAME must be a full 8 characters in length, including spaces as needed. Note that the header printing subroutine used by the Monitor does not print trailing spaces in the file name.

SIZE and ADDRESS indicate the number of bytes and the starting address of the file. They must be formatted as "broken" integers, with the low byte as a string-character-like integer and the high byte as another:

```
(1)-:     (last character of Name)
    ØØLL  (Size)
    ØØHH  (  "  )
    ØØLL  (Address)
    ØØHH  (   "   )
```

Note that this header is not identical with the one used by the Monitor Commands. The actual tape operators have been kept primitive so as to allow latitude when using the operators within applications. The present Monitor is only one of a variety of possible methods of handling the tape/user interface.

WRITE copies the header off the stack and calculates a single-byte checksum that is attached to the end of the header. The carrier signal is then output for about 1Ø dec seconds, followed by the header and then the file itself; finally the carrier is output again for about 3 seconds.

READ expects a string on the stack of either Ø (null string) or 9 bytes. A null string causes READ to input the first file found on tape. A string of 9 bytes should have the desired file type as its first byte, followed by the 8 byte file name (which must include trailing spaces). The file is read into memory at the location pointed to by SYSVAR FIL. READ may terminate in number 6 or 7 TAPE ERROR. READ may aborted by pressing any key. If the file specified was not found, READ returns Ø TOS, if the

file was found and loaded, True is returned, and if aborted 1 is returned.

Note that READ operates regardless of original file length; it is thus possible to overwrite vital parts of memory with an unexpectedly long file.

VER expects the header of the file to be VERified to be in $BUF, exactly as left by WRITE. It then checks each byte against those on tape. If a byte does not match, the process is ended and Ø is returned TOS; if any key is pressed, the process also ends and returns 1 TOS; a successful VERification returns True. Note that other operators also use the $BUF and can disturb any header left there. The Monitor circumvents this by storing the header as a string variable between WRITE and VER.

Additional information regarding the workings of the tape system may be found in Chapter I.

ARRAY OPERATOR

ARRAY remove TOS integer/address; use address contained at TOS address as array base. At array base is a byte denoting the number of dimensions 'n' Ø-ØFF, followed by n bytes of dimensions. N integers on the stack are removed and applied to the dimensions (TOS integer to first dimension, NOS to second, etc.), and the offset of the desired byte element is calculated. The offset added to the address of the first byte element of the array is returned TOS. See page 31 for details on dimensioning an array.

APPENDICES

The following abbreviations are used throughout the Appendices:

x - no operand required
n - hex number Ø-ØFFFF
nn - 2-digit hex byte
nnnn - 4-digit hex integer
  aaaa - address
  llll - length
ss ss - record size] 2
rr rr - runtime addr]bytes
h - hex digit
l - Global Variable A-ZZ
:l - letter variable A-ZZ
     or :A-:Z
t - file type, single char
fn$ - file name, 8 char (1B-3F) or less
... - words/bytes of code
itm - data items
str - string expression
-/- - alternative operands
[] - optional operands
lab - user label, 2 char (ØB-3F) or more

Appendix A MONITOR & EDCOM COMMANDS

Ax - Auto - load and run first file found.
Bx - Debug - link current Link file into test program.
Cx - Compile current Text file.
Daaaa - Dump memory contents from aaaa.
Ex - Edit.
Fx - File Directory - read and print header of each file found.
Gaaaa - Go To - execute RPNZL procedure at aaaa.
Kx - Linker - execute Linker Program.
L[fn$(t)][aaaa] - Load specified file (first found if none specified) into memory at aaaa (at (FIL) if not specified).
Maaaa>aaaa',llll - Move llll bytes from aaaa to aaaa'.
Px - Picture - load and display file SCREEN(S).
Qx - Quit current invocation of RPNZL.
R[aaaa] - Run RPNZL program at aaaa, or at (APPLIC) if not specified.

```
Sfn$(t)aaaa,llll - Save llll bytes from aaaa under
                  name and type fn$(t).
TRx - Transfer Read - load ROM-style file into memory
                      at 65ØØ.
TWllll - Transfer Write - save llll ROM-style bytes
                          from aaaa.
UKx - Kill (erase) current Text file.
ULTfn$ - Load Text file of name specified (8 characters
         must be given).
UPx - Print - send current Text file to printer.
USLx - Save Link file under filename(L)7BØØ,ØlFF.
USTx - Save Text file under filename(T)72EØ,Ø81Ø.
Vx - Verify last file saved.
Xx - Examine Stack - list contents of runtime Stack.
```

Appendix B CONTROL WORDS

```
START nnnn - compile program to start at nnnn.
FORGET x- Erase Dictionary.
STOP x - end of text file; Link Record Type F.

SEGments
PROG n ... END - n=# Global Variables Ø-33; Link
                 Record Type A.
USKEY h SUB ... .FUNK ]- h=key # Ø-F; SUB indicates
        CODE nn itm   ]  RPNZL subroutine; CODE
                         indicates machine code of
                         nn itms.
MOD nn h n ... END - nn=module # ØØ-3l; h=level Ø-7;
                     n=# Local Variables Ø-19; Link
                     Record Type D.
SUB ... END                                  ]
DATA nn itm - nn=# data items                ]  Link
BUF nn nn' itm - nn=# data items;            ]      Record
                 nn'=# blank bytes           ]           Type
$CONS str .                                  ]              E

DEF lab operands and entry types
PWD HERE/nnnn - type Ø          BYT nn - type 1
VAR nl - type 1                 WD HERE/nnnn - type 2
CONS 1 HERE/n  - type 1; Link Record Type C.
```

ENTRY TYPES
Ø - compile Ø2 + value bytes
1 - compile lower value byte
2 - compile both value bytes

LINK RECORD TYPES
A n ss ss rr rr + ssss bytes code
B h * ss ss rr rr + ssss bytes code; *=8Ø SUB, ØØ CODE
C nnnn * - *=variable # Ø-33; no record bytes
D nn ss ss rr rr h n + ssss-2 bytes code
E ss ss rr rr + ssss bytes code
F - no record bytes

Appendix C ERRORS

COMMAND ERRORS
Command Errors are those associated with the Monitor and EDCOM Commands. Most are syntactic and simply cause the Command line to be ignored. However, Verify will generate a message if Verification fails, and Debug will do so if an illegal record is found.

RUNTIME ERRORS
These errors are generated within the language itself. When one occurs, its number is placed in the SYSVAR ERRNUM, and the subroutine pointed to by the SYSVAR ERR is executed. Like the Function Keys, the pointer in ERR has bit 15 set if the subroutine is RPNZL and reset if machine code. Any RPNZL subroutine used must terminate with .FUNK. The present error-handler places the error number (found in ERRNUM) in the BC register to be returned as the value of the BASIC USR function, clears the machine stack, and terminates RPNZL. Since RPNZL may be called recursively, as when the Monitor Runs a program, errors can also be handled by the caller, as is done with the Monitor error messages:

Ø - OK - execution ended with no errors.
1 - BREAK - Break Key Combination (SPACE-Ø) pressed.
2 - STCK ERROR - Parameter Stack underflowed.

3 - NEST ERROR - either the Loop Stack has overflowed, or a subroutine has ENDed inside a loop.

4 - $ EVAL ERR - a string of illegal size has been passed to HEX or DEC.

5 - OUT OF MEM - a string variable assignment has caused SYSVAR FREM to exceed HIMEM.

6 - TAPE ERROR - a framing error has occurred.

7 - TAPE ERROR - file has loaded but checksums do not match

8 - LANG ERROR - the checksum of the System Tape does not match the SYSVAR LANGSUM (4294).

9 - CODE ERROR - an invalid operator code has been found.

COMPILER ERRORS

1 - UNABLE TO COMPILE - an illegal or unexpected word or operand has been found. There are a number of occasions where this may occur:

a) a number is marked - more than 5 characters; starts with character other than Ø-9; Control Word expected.

b) $ is marked - invalid format code; Tab operand over 1F; terminator . missing; should the Compiler appear to "run away" while compiling a string (lots of ØØs), a closing quote may be missing; Control Word expected.

c) Control Word is marked - invalid START operand; previous SEGment not ended properly; wrong number of data items declared.

d) Label/operator is marked - misspelled word; invalid label after DEF; label unDEFined before use; missing / in data Segment; Control Word expected.

2 - DICTIONARY FULL - the last DEF operand is marked in the DEFinition whose entry overflows either Dictionary string,

3 - LOOP ERROR - incorrect end word: END where .FUNK is expected or vice versa; incorrect end-structure: previous end-structure omitted, an incorrect end-structure has been used. Omission of one .CASE in a series of CASE clauses is usually not discovered until

the .SLECT is reached and the imbalance found.
4 - BUFFER FULL - the Link Buffer (7BØØ-7CFF) is full. Usually only happens when a great many strings are used in one Text file; may also indicate a "run away" string (see 1b above).

Omission of a closing parenthesis) can cause portions of source code to be ignored as remarks. Usually undetected until an undefined label or invalid structure is discovered. May cause "run away" compilation if there are no more)s in the Text file.

LINKER ERRORS
ERROR 1 - invalid record or file type in Link file.
ERROR 2 - gap in assignment of module numbers.
ERROR 3 - Global Variable used in DEF 1ab CONS 1 v is outside of range declared by SEG PROG.

Appendix D TRANSLATION FROM SINCLAIR BASIC

An attempt to translate directly from Sinclair BASIC will quickly point out its inherent redundancies in its manipulation of the Parameter Stack. It is usually necessary to redesign code passages to take advantage of the economy of RPNZL's stack operators.

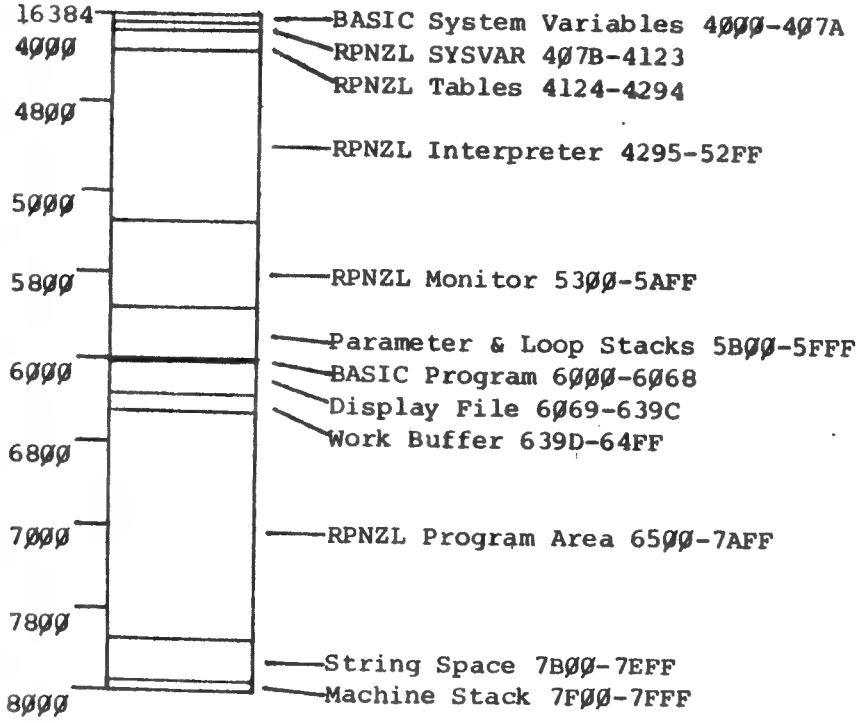
These BASIC keywords govern features not used by RPNZL: ABS ACS ASN ATN CLEAR CONT COS EXP GOTO INT LIST LLIST LN NEW PI SGN SIN SQR STOP TAN **. To varying degrees they may be programmed around.

The following list offers some suggestions for possible translations. Chapter V should be consulted for details of the function of the RPNZL operators.

AND - AND
CHR$ - integer to one byte $ 1
CLS - CLEAR
CODE - of a one byte string
multibyte $ 1 1 /$
COPY - COPY
DIM - SEG BUF nn nn ...
FAST - ØF23 CALL
DROP $ length; of a

FOR...TO...(STEP...)
  NEXT - (STEP)DO...LOOP
IF...THEN - IF...(ELSE
  ...) .IF
INKEY$ - KEY, CHAR
INPUT - (>)INPUT, CHAR
LEN - DUPE
LET - "
LOAD - READ, Cmd L
LPRINT - use Output
  Redirection
NOT - NOT
OR - OR
PAUSE - DELAY
PEEK - PEEK
PLOT/UNPLOT - DOT/UNDOT
POKE - POKE
PRINT (AT) - (>)PRINT, SEND,
  PRINT$, SHAPE,
  POSN
RAND - n 40AC "
REM - ( )
RETURN - END, .FUNK
RND - RAN
RUN - PROG
SAVE - WRITE, Cmd S
SLOW - 0F2B CALL
STR$ - HEX$, DEC$
TAB - within $ Tnn, else
  C0+nn SEND
USR - CALL
VAL - HEX, DEC

Appendix E MEMORY MAP & SYSTEM VARIABLES



SYSVAR - THE RPNZL SYSTEM VARIABLES

```
4Ø7B/C -  IOSTAT - flags
4Ø7D-85 - header of BASIC program Line 1
4Ø86/7 -  MON - address of Monitor
4Ø88-F -  RCPT, RSPT, RLSP, RPSBOT: runtime pointers
4Ø9Ø/1 -  CPT - Code Pointer (program counter)
4Ø92/3 -  SPT - Parameter Stack Pointer
4Ø94/5 -  LSP - Loop Stack Pointer
4096/7 -  APPLIC - address of current program
4Ø98-B -  PSBOT, LSBOT - stack bases
4Ø9C-F -  GLOBVTBL, LOCVTBL - addresses of current
          Variable Tables
4ØAØ/1 -  MODTBL - address of current Module Table
4ØA2/3 -  not used
4ØA4-7 -  FREM, HIMEM - limits of string space
4ØA8/9 -  $BUF - address of string work buffer
4ØAA/B -  T$LEN, N$LEN - work variables for strings
4ØAC/D -  RND - pseudorandom seed
4ØAE/F -  OVFW - carry, borrow, overflow, or remainder
          from + - * / LSHIFT RSHIFT
4ØBØ-3 -  INDEX, LIMIT - DO loop control variables
4ØB4/5 -  SLCT - search key for SELECT
4ØB6/7 -  ERR - address of error handler subroutine
4ØB8/9 -  STP - DO loop Step
4ØBA-C -  not used
4ØBD -    ERRNUM - error report code
4ØBE/F -  FIL - address of current file
4ØCØ-3 -  PRPOS, PRADDR - current print position
          (inverted) and address
4ØC4-7 -  COORD, XSIZE, YSIZE - control variables for
          FIG
4ØC8-B -  BCURS, ACURS, MODCH, MAXIN - control
          variables for INPUT
4ØCC/D -  CTBL - address of Character Table
4ØCE/F -  INPTR - input buffer pointer
4ØDØ/1 -  FKTBL - address of current Function Key Table
4ØD2-6 -  EVALBUF - work space for string evaluation
4ØD7 -    not used
4ØD8-4123 - USR1-USR38 - user variables 1-38 dec
:
4294 -    LANGSUM - System Checksum
```

Appendix F MACHINE CODE

RPNZL allows machine code within RPNZL programs in a first class fashion by use of the CALL operator in conjunction with the data Segment. CALL removes the TOS integer/address and executes machine code at that address. It does not automatically return a value as the BASIC USR function does.

It is possible, however, to access the Parameter Stack from machine code by means of CALL STACK (CD 53 43), which puts the contents of the DE register pair onto the Parameter Stack, and CALL DROP (CD 64 43), which pops the TOS integer into DE.

Additionally, CALL RPNZLRT (CD AB 4D) will reinvoke RPNZL to execute a RPNZL procedure at the address in DE, provided the procedure ends with .FUNK.

SEG DATA nn itm should be used to include machine code, with its address defined by DEF lab PWD HERE. Use of word-value and byte-value labels within the data Segment allows expression of op-code operands in symbolic form. This is done by prefixing / to the label in question. Study the following Segments:

```
DEF BLOB BYT 8Ø    (some labels)
DEF DFILE WD 4ØØC (to use in m/c)
DEF BLOBIT PWD HERE
SEG DATA Ø6
 21 /DFILE          (LD HL, DFILE)
 23                 (INC HL)
 36 /BLOB           (LD ;HL;,BLOB)
 C9                 (RET)
SEG SUB ... BLOBIT CALL ... END
```

It is also possible to modify the TTBL at 4124 by vectoring unused codes to machine code subroutines.

Fast ~~0F27~~ 0F23
Slow 0F2B

Appendix G CODE SUMMARY

	00	END	4	20	XOR		40	SEND	60	KEY
	01	(byte)	5	21	=		41	PRINT	61	CHAR
	02	(integer)	6	22	<		42	POSN	62	INPUT
	03	$	7	23	>		43	CLEAR	63	>INPUT
	04	1	8	24	BUMP		44	HOME	64	.FUNK
	05	2	9	25	<=		45	>PRINT	65	COPY
	06	TRUE	A	26	>=		46	UP	66	OPEN
	07	0	B	27	INDX		47	DOWN	67	CLOSE
	08	SWAP	C	28	OVF		48	CR	68	WRITE
	09	DUPE	D	29	FUN		49	$PRINT	69	READ
	0A	DROP	E	2A	CALL		4A	CELL	6A	VER
	0B	NTH	F	2B	PROG		4B	SHAPE	6B	FILE
	0C	ROT	G	2C	"CP		4C	DOT		
	0D	OVER	H	2D	PROC		4D	UNDOT	6C-7E	n/u
	0E	n/u	I	2E	LEAVE		4E	BIT		
	0F	NEW	J	2F	WHILE		4F	FIG		
	10	£	K	30	IF		50	RAN		
	11	"	L	31	BEGIN		51	HEX		
>	12	PEEK	M	32	THEN		52	DEC		
<	13	POKE	N	33	DO		53	HEX$		
=	14	SWITCH	O	34	SELECT		54	DEC$		
+	15	DELAY	P	35	CASE		55	ARRAY		
-	16	RSHIFT	Q	36	STEPDO		56	IN$		
*	17	LSHIFT	R	37	n/u		57	SWAP$		
/	18	+	S	38	ELSE		58	"$		
;	19	-	T	39	.IF		59	£$		
,	1A	*	U	3A	UNTIL		5A	AS$		
.	1B	/	V	3B	WEND		5B	+$		
0	1C	NOT	W	3C	LOOP		5C	/$		
1	1D	NEG	X	3D	.CASE		5D	=$		
2	1E	AND	Y	3E	.SLECT		5E	DROP$		
3	1F	OR	Z	3F	n/u		5F	DUPE$	7F	NOP

80-B1 - :00 through :31 - Modules (Mod #, lev, #var)

B2-CB - :A through :Z - Local Variables

CC-FF - A through ZZ - Global Variables

CH
DAT

WARRANTY

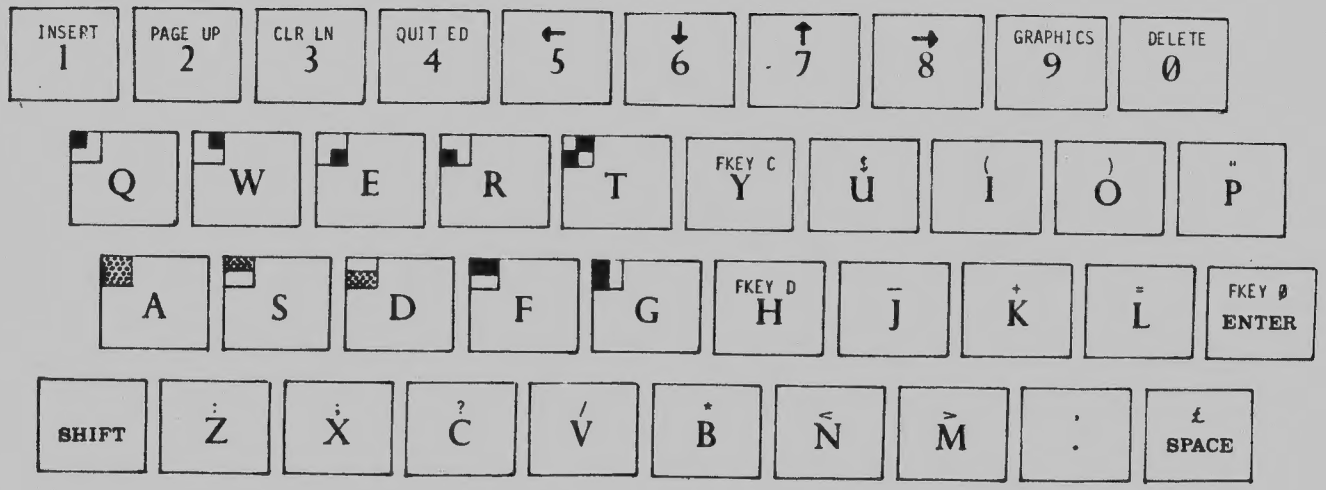
This Software Package is warranted to the owner for a period of 90 days from date of purchase against defects in media. If either tape is discovered to be unloadable within 90 days from date of purchase, The Golden Stair will replace the defective tape(s).

The Golden Stair is not liable for any incidental or consequential damage arising from use of the Software Package or of any program(s) developed by means of or in conjunction with the Package.

LICENSING POLICY

The RPNZL Programming System is protected by U.S. Copyright Law. The Golden Stair believes that the success of a system such as this depends upon its wide distribution, and there encourages the making of copies of the System for backup purposes and to give to friends, clubs, etc. We only ask that the user not make copies for sale. Programs developed on the System require no permission other than acknowledgement of The Golden Stair for their sale or distribution.

BB



CHARACTERS ABOVE ARE REGULAR MODE.
GRAPHICS MODE CHARACTERS ARE THE EXACT INVERTS OF THE ABOVE.

FUNCTION KEYS

Key Ø	shifted ENTER	Key 4	shifted 4	Key 8	shifted 8	Key C	shifted Y
1	" 1	5	" 5	9	" 9	D	" H
2	" 2	6	" 6	A	" Ø	E	n/u
3	" 3	7	" 7	B	unshifted Enter	F	n/u